6장 동적 계획법

목차

- 기본 개념
  - 동적 계획법
  - 분할 정복과의 비교
  - 최적성의 원리
- 행렬의 연쇄적 곱셈
  - 직선적 알고리즘
  - 동적 계획법을 사용한 알고리즘
- 최적 이진 탐색 트리
- 스트링 편집 거리

기본 개념

- 동적 계획법(dynamic programming)
  - 주어진 문제를 여러 개의 소문제로 분할하여 각 소문제의 해결안을 바탕으로 주어진 문제를 해결
  - 각 소문제는 다시 또 여러 개의 소문제로 분할 가능
  - 각 소문제는 원래 주어진 문제와 동일한 문제이지만 입력의 크기가 작음
  - 소문제의 해를 표 형식으로 저장해 놓고 이를 이용하여 입력 크기가 큰 원래의 문제를 점진적으로 해결

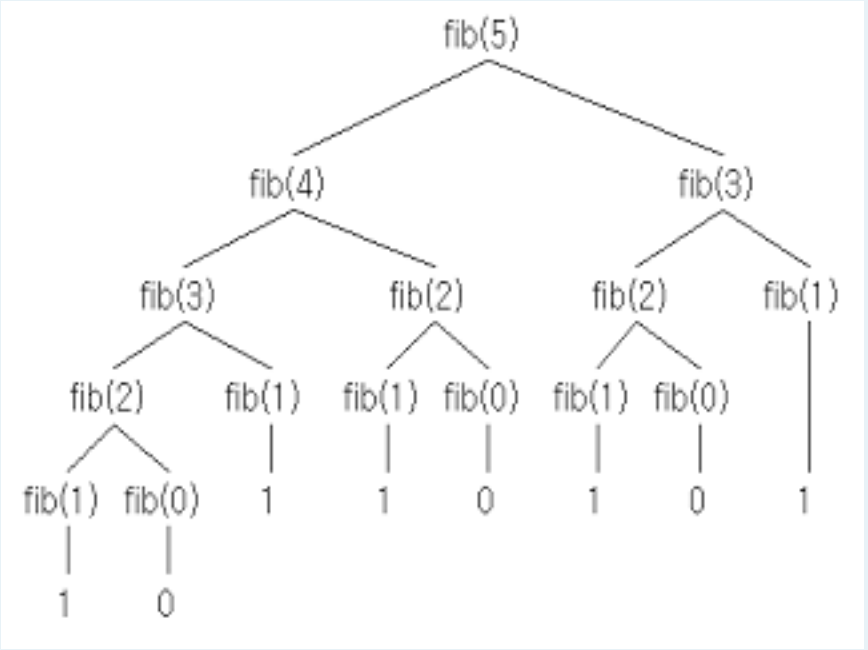
분할 정복과의 비교

- 분할 정복
  - 분할되는 소문제가 독립적이어서 소문제를 다시 순환적으로 풀어 그 결과를 합침
- 동적 계획법
  - 소문제가 독립적이지 않아서, 즉 소문제 간에 중복되는 부분이 있어서 이를 분할 정복 방법으로 풀면 동일한 소문제를 반복적으로 풀어야 하는 경우가 발생
  - 소문제의 계산 결과를 표에 저장해 놓고 필요할 때 이 표에서 값을 꺼내옴

분할 정복과의 비교 예(1)

- 피보나치 수열
  - $f(n) = f(n\text{-}1) + f(n\text{-}2)$
- 분할 정복의 경우
  - 아래 함수를 반복 호출

```
def fib(n):
    if n <= 0: return 0
    if n == 1: return 1
    else: return fib(n-1) + fib(n-2)
```



분할 정복과의 비교 예(2)

- 동적 계획법의 경우
  - f[2]부터 f[3], f[4]의 순서로 계산하여 f[n]을 구함
  - 상향식 방법 (bottom-up)

```
def fib2(n):
    f = [0] * n
    if n > 0:
        f[1] = 1
        for i in range(2, n):
            f[i] = f[i-1] + f[i-2]
    return f[n-1]
```

적용 대상

- 최적화 문제
  - 동적 계획법은 최소치 또는 최대치를 구하는 최적화 문제에 적용
  - 최적화 문제의 최적해는 여러 개가 있을 수 있지만 그 중의 어느 하나를 구하면 됨
  - 최적해는 기본적인 소문제의 최적해로부터 더 큰 크기의 소문제의 최적해를 구하는 과정을 거침
- 최적성의 원리(principle of optimality)
  - 주어진 문제의 부분의 해가 전체 문제의 해를 구성하는데 사용
  - 동적 계획법으로 문제를 해결하려면, 그 문제가 최적성의 원리를 만족해야 함
- 최적성의 원리의 적용
  - 어떤 문제의 한 사례(instance)에 대한 최적해가 모든 부분사례(sub-instance)에 대한 최적해들을 항상 포함할 때

적용 단계

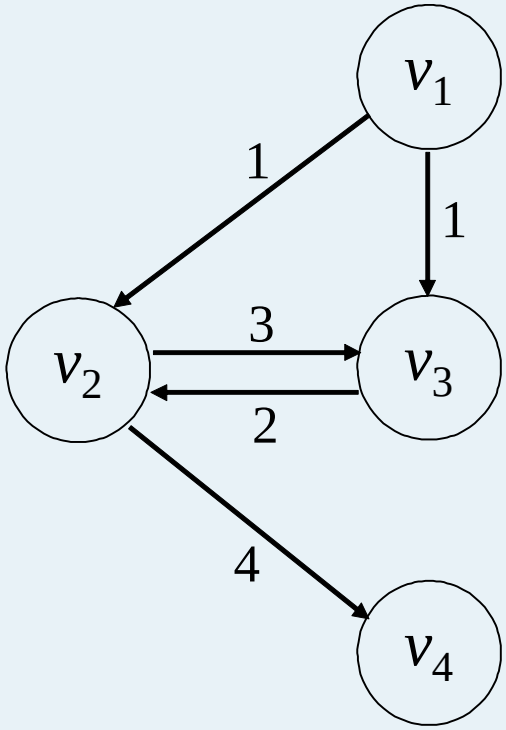
① 문제의 특성을 분석하여 최적성의 원리가 적용되는지 확인

② 주어진 문제를 소문제로 분해하여 최적해를 제공하는 점화식 도출

③ 입력 크기가 작을 때 도출된 점화식의 해를 구함

④ 이 해를 이용하여 점차적으로 입력 크기가 클 때의 점화식의 최적해를 구함

최적성의 원리가 성립하지 않는 예



- v_1에서 v_4까지의 최장 경로 : $[v_1, v_3, v_2, v_4]$

- v_1에서 v_3까지의 최장 경로 : $[v_1, v_3]$가 아니라 $[v_1, v_2, v_3]$

- 부분 해가 전체 해를 구성하는데 사용되지 않음

욕심쟁이 방법과의 비교

- 두 방법 모두 최적성의 원리가 적용
- 욕심쟁이 방법 (greedy method)
  - 단계별로 진행되어 각 단계에서는 현재 상태에서 가장 최적이라고 판단되는 결정
  - 국부적인 최적해들이 결국에는 전체적인 최적해로 된다는 전략
  - 소문제에 대한 하나의 최적해만을 고려
  - 문제에 따라 최적해를 구할 수 있고, 그렇지 않을 수도 있음
- 동적 계획법
  - 소문제에 대한 여러 최적해로부터 다음 크기의 소문제에 대한 최적해가 결정
  - 모든 가능성을 고려하여 결정을 내리게 되므로 항상 최적의 결과

행렬의 연쇄적 곱셈(1)

- n 개의 행렬을 곱함
  - $M_1 \times M_2 \times \ldots \times M_n$
  - $(i \times j$ 행렬$) \times (j \times k$ 행렬$) = (i \times k$ 행렬$)$
- 결합법칙의 성립
  - 행렬을 곱하는 순서에 따라 여러 가지 다른 방법으로 계산
- 알고리즘의 목표
  - 여러 개의 곱셈 순서 중에서 비용이 최소인 곱셈 순서를 구하는 것

행렬의 연쇄적 곱셈(2)

- n = 4, M_1부터 M_4까지의 차원이 각각 10×100, 100×5, 5×50, 50×20이라고 할 때 이것들을 곱하는 방법

① $M_1 \times (M_2 \times (M_3 \times M_4))$: 5×50×20 + 100×5×20 + 10×100×20 = 35,000

② $M_1 \times ((M_2 \times M_3) \times M_4)$: 100×5×50 + 100×50×20 + 10×100×20 = 145,000

③ $(M_1 \times M_2) \times (M_3 \times M_4)$: 10×100×5 + 5×50×20 + 10×5×20 = 11,000

④ $((M_1 \times M_2) \times M_3) \times M_4$: 10×100×5 + 10×5×50 + 10×50×20 = 17,500

⑤ $(M_1 \times (M_2 \times M_3)) \times M_4$: 100×5×50 + 10×100×50 + 10×50×20 = 85,000

행렬의 연쇄적 곱셈(3)

- 직선적(brute-force) 알고리즘
  - 가능한 모든 순서를 모두 고려해 보고, 그 가운데에서 곱셈 횟수가 가장 최소인 것을 택하는 것
  - n 개의 행렬$(M_1, M_2, \ldots, M_n)$을 곱할 수 있는 모든 순서의 가지 수를 t_n 이라 할 때
    - $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2$
      $= 2^{n-2} = O(2^n)$

행렬의 연쇄적 곱셈(4)

- 동적 계획법을 사용한 알고리즘
  - n 개의 행렬을 곱할 때 n-1 번의 곱셈 필요
  - n 개의 행렬 중 M_i와 M_{i+1}, $1 \le i \le n-1$ 을 곱함
  - 두 개의 행렬이 하나의 새로운 행렬로 대체
  - 이제 n-1 개의 행렬을 곱하는 문제로 변환
  - 이러한 작업을 계속하면 마지막에는 두 개의 행렬이 남고 이들을 곱함

행렬의 연쇄적 곱셈(5)

- 점화식
  - $1 \le i \le j \le n$
  - $M[i,j]$: $i < j$ 일 때, A_i부터 A_j까지의 행렬을 곱하는데 필요한 기본적인 곱셈의 최소 횟수
  - $M[i,j] = \min_{i \le k \le j-1}(M[i,k] + M[k+1,j] + d_{i-1}d_k d_j)$, if $i < j$
  - $M[i,j] = 0$, if $i \ge j$

행렬의 연쇄적 곱셈 알고리즘

```
matrixChainMult(d[], p[], n)
  for (i ← 1; i ≤ n; i ← i + 1) do
    M[i,i] ← 0;
  for (h ← 1; h ≤ n−1; h ← h + 1) do
    for (i ← 1; i ≤ n−h; i ← i + 1) do {
      j ← i + h;
      M[i,j] ← min_{i≤k≤j−1}(M[i,k] + M[k+1,j] + d[i−1]·d[k]·d[j]);
      p[i,j] ← 최소 값을 갖는 k;
    }
  return M[1,n];
end matrixChainMult()
```

행렬 곱셈의 예

A	B	C	D	E	F
4 × 2	2 × 3	3 × 1	1 × 2	2 × 2	2 × 3

풀이

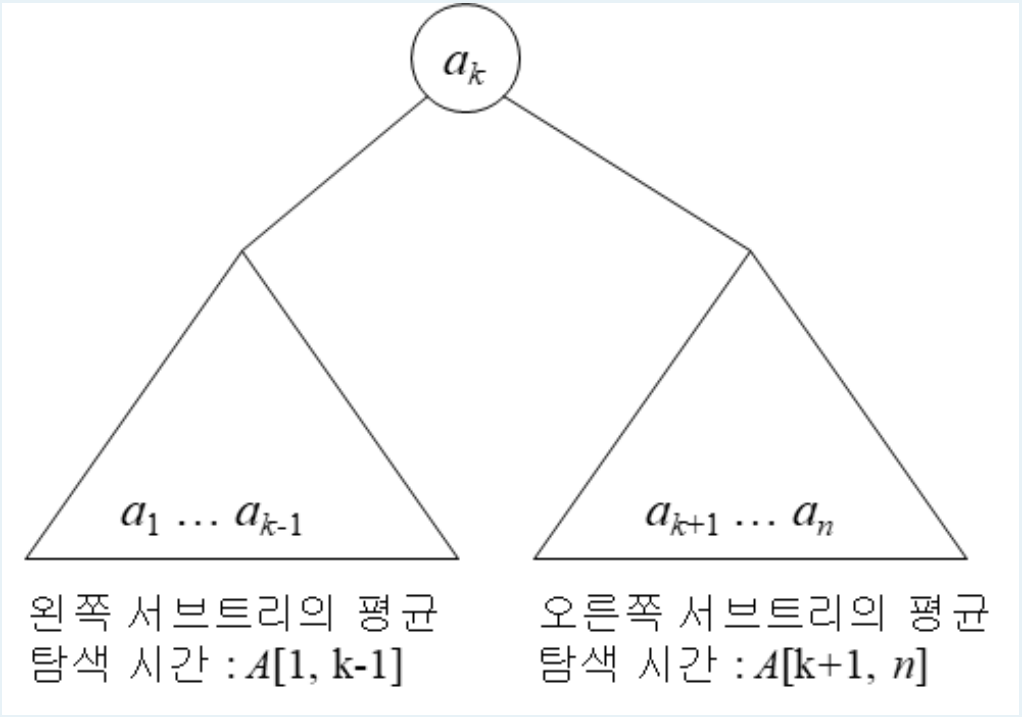
	B	C	D	E	F
A	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC][DEF]
C			6 [C][D]	10 [C][DE]	19 [C][DEF]
D				4 [D][E]	10 [DE][F]
E					12 [E][F]

최적 이진 탐색 트리(1)

- 이진 탐색 트리
  - 루트의 왼쪽 서브트리에 있는 원소의 키 값은 루트보다 작고, 루트의 오른쪽 서브트리에 있는 원소의 키 값은 루트보다 큰 이진 트리
- 최적 이진 탐색 트리
  - 트리 내의 키와 각 키가 탐색될 확률이 주어져 있을 때 그 트리의 평균 탐색 비용, 즉, 평균 비교 횟수를 계산하고 이를 최소화하는 탐색 트리를 구축하는 문제

최적 이진 탐색 트리(2)

- 키 값 $a_i \le a_{i+1} \le \ldots \le a_j$ 일 경우 $A[i,j]$는 이진 탐색 트리의 i부터 j까지의 노드에 대한 최소 평균 탐색 시간



최적 이진 탐색 트리(3)

- 점화식

$$A[i, j] = \min_{i \le k \le j}(A[i, k-1] + A[k+1, j] + \sum_{q=i}^{j} P(a_q))$$

$$A[i, i] = P(a_i)$$

$$A[i, i-1] = 0, 1 \le i \le n-1$$

최적 이진 탐색 트리 구하기(1)

- a_1 = A, a_2 = B, a_3 = C, a_4= D
- A < B < C < D
- p_1 = 0.3, p_2 = 0.2, p_3 = 0.4, p_4 = 0.1

최적 이진 탐색 트리 구하기(2)

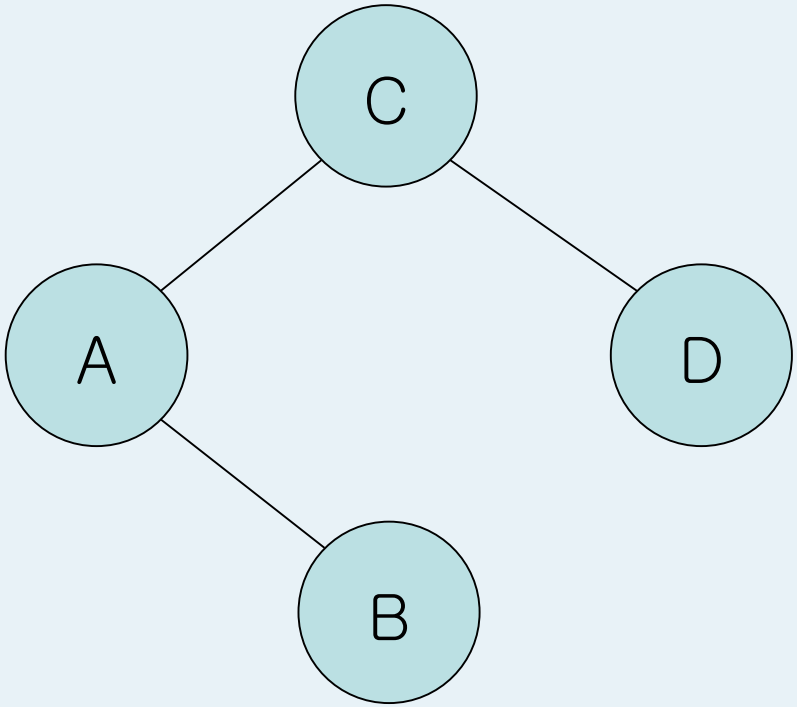
$A[i,j]$의 값

i \ j	1	2	3	4
1	0.3	0.7	1.6	1.8
2		0.2	0.8	1.0
3			0.4	0.6
4				0.1

최소값을 갖는 k의 값

i \ j	1	2	3	4
1	1	1	2	3
2		2	3	3
3			3	3
4				4

최적 이진 탐색 트리 구하기(3)



최적 이진 탐색 트리 알고리즘

```
optimalBST(p[], r[], n)
  for (i ← 1; i ≤ n; i ← i + 1) do {
      A[i,i] ← p[i];
      r[i,i] ← i;
  }
  for (h ← 1; h < n; h ← h + 1) do
      for (i ← 1; i ≤ n-h; i ← i + 1) do {
          j ← i + h;
          A[i,j] ← min_{i≤k≤j}(A[i,k-1] + A[k+1,j] + Σ P[m]);
          r[i,j] ← 최소 값을 갖는 k;
       }
  return A[1,n];
end omtimalBST()
```

스트링 편집 거리(1)

- 스트링 편집 거리(string edit distance)
  - 두 스트링의 유사도를 측정하기 위해 사용
  - Levenshtein distance(LD)라고도 함
  - 원래 스트링을 S, 목표 스트링을 T
  - S를 T로 변환하는데 필요한 삽입, 삭제, 대치 연산의 최소 비용
  - 편집 거리가 커질수록, 두 스트링의 유사도는 낮아지게 됨
    - 논문이나 보고서의 표절 검사, DNA 염기 서열의 유사도 검사 등에 사용됨

스트링 편집 거리(2)

- 점화식
  - 삽입 연산의 비용 : δ_I
  - 삭제 연산의 비용 : δ_D
  - 대치 연산의 비용 : δ_S
  - $D[i,j] : S = s_1 s_2 \ldots s_i$와 $T = t_1 t_2 \ldots t_j$ 사이의 편집 거리
  - $D[i,j] = \min(D[i,j\text{-}1] + \delta_I, D[i\text{-}1,j] + \delta_D, D[i\text{-}1,j\text{-}1] + 0/\delta_S)$
    - 여기서 $0/\delta_S$는 $s_i = t_j$이면 0이고, 그렇지 않으면 δ_S임을 의미

스트링 편집 거리(3)

- 동적 계획법의 적용
  - 위의 점화식에 의하면 편집 거리에 대하여 최적성의 원리가 성립
  - $D[i,j]$를 단순히 순환적으로 계산하면 중복될 수 있으므로 동적 계획법을 적용
  - $\delta_I = \delta_D = \delta_S = 1$인 경우의 점화식
    - $D[i,j] = \min(D[i,j\text{-}1] + 1, D[i\text{-}1,j] + 1, D[i\text{-}1,j\text{-}1] + 0/1)$

스트링 편집 거리 예

- $\delta_I = \delta_D = \delta_C = 1$ 일 때,

  S = GUMBO 를

  T = GAMBOL 로 변경하는 스트링 편집 거리

풀이(1)

		G	U	M	B	O
	0	1	2	3	4	5
G	1					
A	2					
M	3					
B	4					
O	5					
L	6					

풀이(2)

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1				
M	3	2				
B	4	3				
O	5	4				
L	6	5				

풀이(3)

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

스트링 편집 거리 알고리즘

- $\delta_I = \delta_D = \delta_S = 1$인 경우 알고리즘

```
editDistance(s[], t[], m, n)
  // 문자 배열 s[1,m], t[1,n]
  D[0,0] ← 0;
  for (i ← 1; i ≤ n; i ← i + 1) do
    D[i,0] ← D[i-1,0] + 1;
  for (j ← 1; j ≤ m; j ← j + 1) do
    D[0,j] ← D[0,j-1] + 1;
  for (i ← 1; i ≤ n; i ← i + 1) do
    for (j ← 1; j ≤ m; j ← j + 1) do {
      if (s[i] = t[j]) then cost ← 0;
      else cost ← 1;
      D[i,j] ← min(D[i,j-1] + 1, D[i-1,j] + 1, D[i-1,j-1] + cost);
    }
  return D[n,m];
end editDistance()
```